MCDB-R: Risk Analysis in the Database

Subi Arumugam[1]
Fei Xu[3,*]

Ravi Jampani[1]
Christopher Jermaine[2]

Luis L. Perez[2]
Peter J. Haas[4]

[1]University of Florida
Gainesville, FL, USA
{sa2,jampani}@cise.ufl.edu

[2]Rice University
Houston, TX, USA
{lp6,cmj4}@rice.edu

[3]Microsoft Corporation
Redmond, WA, USA
feixu@microsoft.com

[4]IBM Research - Almaden
San Jose, CA, USA
phaas@us.ibm.com

ABSTRACT

Enterprises often need to assess and manage the risk arising from uncertainty in their data. Such uncertainty is typically modeled as a probability distribution over the uncertain data values, specified by means of a complex (often predictive) stochastic model. The probability distribution over data values leads to a probability distribution over database query results, and risk assessment amounts to exploration of the upper or lower tail of a query-result distribution. In this paper, we extend the Monte Carlo Database System to efficiently obtain a set of samples from the tail of a query-result distribution by adapting recent "Gibbs cloning" ideas from the simulation literature to a database setting.

1. INTRODUCTION

In the face of regulatory processes such as Basel II and Solvency 2, enterprises are becoming increasingly concerned with managing and assessing the credit, financial, engineering, and operational risk arising from uncertain data [16]. Examples of uncertain data include future values of financial assets, customer order quantities under hypothetical price changes, and transportation times for future shipments under alternative shipping schemes.

Data uncertainty is usually modeled as a probability distribution over possible data values, and such probabilities are often specified via complex stochastic models. E.g., we might specify the foregoing uncertain financial, retail, and logistics data sets using Euler approximations to stochastic differential equations, Bayesian demand models, and stochastic network models, respectively. This specification in turn leads to the representation of data uncertainty as a probability distribution over possible database instances ("possible DBs," also called "possible worlds"). Running an aggregation query such as "select sum of sales" over uncertain data therefore does not yield a single, deterministic result for total sales; rather, there is a probability distribution over all possible query results.

In this setting, risk assessment typically corresponds to computing interesting properties of the upper or lower tails of the query-result distribution; for example, computing the probability of a large investment loss. The problem is made more complex by the fact that the tails of the distribution are hard to specify a priori, and so risk analysis frequently focuses on the inverse problem of determining an extreme quantile, e.g., determining the value γ such that there is a 0.1% probability of seeing a loss of γ or more. Such a "value at risk" can be viewed as a probabilistic worst-case scenario, and might be used to specify precisely where the "upper tail" of the loss distribution starts. Proceeding further, one might be interested in computing a more sophisticated "coherent" risk measure such as "expected shortfall" [16], which in this example is defined as the expected total loss, given that this loss exceeds γ. More generally, the entire conditional distribution of the loss—given that the loss exceeds γ—might be of interest.

[*]Fei Xu performed this work while at U. Florida.

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

MCDB In prior work, the Monte Carlo Database System (MCDB) of Jampani et al. [13] was designed for flexible exploration of query-result distributions under arbitrary SQL queries and a broad range of complex, user-defined stochastic models. MCDB uses (possibly user-defined) "variable generation" (VG) functions to pseudorandomly generate instances of each uncertain data value in a database, yielding a sample from the possible-DB distribution. Repeating this process multiple times (i.e., executing multiple "Monte Carlo repetitions") generates a set of independent and identically distributed (i.i.d.) samples from this distribution. Given an SQL aggregation query of interest, MCDB executes the query on each sampled DB instance, thereby generating i.i.d. samples from the query-result distribution. MCDB uses Monte Carlo techniques to estimate interesting features of the query-result distribution—the expected value, variance, and quantiles of the query answer—along with probabilistic error bounds on the estimates. Importantly, a VG function takes as input one or more *parameter tables* (ordinary relations) that control the function's behavior, and produces as output a table containing one or more correlated data values.

For n Monte Carlo repetitions, MCDB does not actually materialize an uncertain database n times; the costs for such a naive approach would be exorbitant. Instead, MCDB executes a query plan only once over a set of *tuple bundles* rather than over ordinary tuples—no matter how many Monte Carlo repetitions are required—often leading to significant time savings. A tuple bundle encapsulates the instantiations of a tuple over a set of generated DB instances, and carries along the pseudorandom number seeds used by the VG functions to instantiate the uncertain data values.

Limitations of MCDB for Risk Analysis Unfortunately, naive Monte Carlo, as implemented in the original MCDB prototype, is not the best tool for exploring the tails of a query-result distribution. Consider the query `SELECT SUM(loss) AS totalLoss FROM t`, where `t.loss` is an uncertain attribute, perhaps representing a future financial loss. Suppose that the query-result distribution is normally distributed, with a mean of \$10 million and a standard deviation of \$1 million. Suppose that interest centers on the upper tail of this distribution that corresponds to `totalLoss` values of \$15 million or more. On average, roughly 3.5 million Monte Carlo repetitions are required before such an extremely high loss is observed even once. If our goal is simply to estimate the area of the tail (i.e., the probability of seeing a `totalLoss` value of \$15 million or more), then 130 billion repetitions are required to estimate the desired probability to within $\pm 1\%$ with a confidence of 95%. If the user instead wants to define the tail indirectly starting at the value γ, where γ is the 0.999 quantile of the `totalLoss` distribution, then standard quantile-estimation techniques [19, Sec. 2.6] require roughly ten million Monte Carlo repetitions to estimate γ to within $\pm 1\%$ with a confidence of 95%,

Our Contributions In this paper, we describe an enhanced version of MCDB for risk analysis called *MCDB-R*. MCDB-R inherits the key strengths of MCDB. Chief among these is generality. MCDB can handle almost any stochastic model, as well as database operations such as aggregation, grouping, and so forth. However, unlike MCDB, MCDB-R supports efficient risk analysis. This presents several technical challenges: efficiently locating a tail of interest and efficiently sampling from the tail, all in the presence of black-box VG functions and complex SQL queries. Our contributions are as follows.

1. We adapt "cloning" and Gibbs-sampling ideas from the rare-event simulation literature to develop a statistical method for both estimating a user-specified quantile on a query-result distribution and then generating a set of samples from the tail.

2. We show how the tail-sampling method can be integrated into MCDB's current tuple-bundle processing infrastructure.

3. In the Appendix, we provide guidance on both setting the sampling parameters for MCDB-R and identifying situations in which MCDB-R will be effective.

Related Work The original MCDB system was partially inspired by several efforts to push analytic capabilities into the database or "close to the data," such as MauveDB [10], which was oriented toward sensor data and integrated smoothing and interpolation methods. These ideas have been further developed in subsequent systems such as FunctionDB [21] and the SciDB project [20]. A recent discussion of analytics in the database can also be found in [6], and there have been a number of efforts to push statistical analyses into Map-Reduce data processing environments; see, e.g., [5, 12].

MCDB is also related to "probabilistic databases" (PrDBs) [7, 8]. Uncertainty in a PrDB is typically represented not by general stochastic models as in MCDB, but rather by explicit probabilities on alternative tuple values. MCDB can capture the functionality of a PrDB, but will not be as efficient or precise in cases where a PrDB can compute answers exactly and efficiently. Overall, MCDB is quite different from PrDBs in terms of motivation, design, and application space. The recent PIP system [14] combines PrDB and Monte Carlo techniques, yielding superior performance for certain MCDB-style queries with simple VG-function parameterizations.

The current paper is also related to the general problem of "conditioning" a probabilistic database, as discussed in Koch and Olteanu [15]. The work in [15] focuses on exact probability calculations in the setting of PrDBs, whereas the current paper emphasizes Monte-Carlo-based approximations.

2. SPECIFYING QUERIES IN MCDB-R

In this section, we indicate what MCDB-R looks like to a user, via a simple example. Suppose that we want to assess potential total financial loss over a specified subset of our clients. The steps in the analysis are to (1) define a stochastic loss model, (2) execute a `SUM` query over the uncertain loss values specified by the model, and then (3) report pertinent features of the total-loss distribution.

For step (1), the user defines an uncertain loss table, just as in MCDB. Suppose for simplicity that we "model" the loss for a customer as being distributed according to a normal distribution with variance 1 and a mean that is customer specific. For this simple model, we use the built-in `Normal` VG function that generates normally distributed random numbers. The data needed to parameterize this function is stored on disk in a "parameter table" (an ordinary SQL table) called `means(CID,m)`, which stores the mean loss for each customer. The uncertain table `Losses(CID,val)` that we wish to query is not stored on disk; only the schema is stored. The schema is specified by the following SQL-like statement:

```
CREATE TABLE Losses (CID,val) AS
  FOR EACH CID IN means
    WITH myVal AS Normal(VALUES(m, 1.0))
    SELECT CID, myVal.* FROM myVal
```

This statement is identical to a standard SQL `CREATE TABLE` statement, except for the `FOR EACH` clause. The statement gives a recipe for constructing a sample instance of `Losses`. Scan through the CIDs in the `means` table. For each CID, create a one-row, one-column table `myVal` by invoking the `Normal` VG function, parameterized with the mean loss for the CID and the default variance value of 1.0. Form a row of the DB instance by joining `myVal` with the CID from the outer `FOR EACH` loop, as specified in the final `SELECT` clause. See [13] for more details on schema specification.

For steps (2) and (3), we write a query as follows:

```
SELECT SUM(val) as totalLoss
FROM Losses
WHERE CID < 10010
WITH RESULTDISTRIBUTION MONTECARLO(100)
  DOMAIN totalLoss >= QUANTILE(0.99)
    FREQUENCYTABLE totalLoss
```

The first three lines correspond to a standard SQL (aggregation) query. Since this query is being executed over uncertain data, there is a probability distribution over the query result. The `DOMAIN` clause indicates that we want to condition the query-result distribution: we discard all possible query results not in the specified domain and then we renormalize the query-result distribution so that the total probability over the domain is 1. In our example, the domain coincides with the upper tail of the loss distribution corresponding to the highest 1% of losses. The `MONTECARLO` keyword indicates that we will use an approximate (conditioned) query-result distribution estimated from 100 Monte Carlo samples.

The remaining lines of the `WITH` clause specify the statistical features of the query-result distribution that are to be computed. The `FREQUENCYTABLE` keyword causes creation of an ordinary table `FTABLE(totalLoss,FRAC)` in which the first column contains the distinct values of `totalLoss` observed over the 100 tail samples, and `FRAC` is the fraction of Monte Carlo samples in which each value was observed. By running the query

```
SELECT MIN(totalLoss) FROM FTABLE
```

we obtain an estimate of the lower tail-boundary, i.e., the extreme quantile of interest, which is accurate when the number l of Monte Carlo samples is large.[1] Similarly, by executing a query such as

[1]As discussed in the next section, our tail-sampling algorithm also produces an estimate of the extreme quantile, but the two estimates will be almost identical for large l.

Algorithm 1 Systematic Gibbs sampler

```
 1: Inputs:
 2:     $X^{(0)}$: initial random element of $\mathcal{X}^r$
 3:     $k$: number of Gibbs updating steps
 4: Output:
 5:     $X^{(k)}$: updated value of $X^{(0)}$
 6:
 7: GIBBS($X^{(0)}, k$):
 8: $x \leftarrow X^{(0)}$                                    // Initialize
 9: for $j \leftarrow 1$ to $k$ do
10:     // Perform one systematic updating step: $X^{(j-1)} \to X^{(j)}$
11:     for $i \leftarrow 1$ to $r$ do
12:         $x_i \leftarrow$ GENCOND($x_{-i}, i$)    // Generate from $h_i^*(\cdot\,|x_{-i})$
13:     end for
14: end for
15: Return $x$                                          // $= X^{(k)}$
```

```
SELECT SUM(totalLoss * FRAC) FROM FTABLE
```

we can estimate the expected shortfall, i.e., the expected loss, given that the loss was among the highest 1% of losses.

3. MONTE CARLO BACKGROUND

We next review results from the Monte Carlo literature and show how they are relevant to the problem of estimating an extreme quantile and efficiently sampling from the tail defined by the quantile. The key methods that we adapt are Gibbs sampling and cloning-based methods for rare-event simulation.

3.1 Gibbs Sampling

Gibbs sampling [11] is a technique that is designed for generating samples from a high-dimensional probability distribution function that is known only up to a normalizing constant. For simplicity of notation, we describe the method in the setting of simple discrete random variables; the technique extends straightforwardly to continuous and mixed random variables. Let $X = (X_1, X_2, \ldots, X_r)$ be an r-dimensional random vector taking values in a discrete set $\mathcal{X}^r$ and distributed according to h, so that $h(x_1, \ldots, x_r) = P(X_1 = x_1, \ldots, X_r = x_r)$ for $x = (x_1, \ldots, x_r) \in \mathcal{X}^r$. A key requirement of Gibbs sampling is that we can efficiently generate samples from the conditional distributions

$$h_i^*(u|v) = P(X_i = u \mid X_j = v_j \text{ for } j \neq i),$$

where $1 \leq i \leq r$. According to this definition, h_i^* is the marginal distribution of X_i, given the values of X_j for $j \neq i$. For a vector $x \in \mathcal{X}^r$, we set $x_{-i} = (x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_r)$ and then we can define h and h_i^* more concisely as $h(x) = P(X = x)$ and $h_i^*(u|v) = P(X_i = u \mid X_{-i} = v)$ for $u \in \mathcal{X}$ and $v \in \mathcal{X}^{r-1}$.

Given an initial value $X^{(0)} = (X_1^{(0)}, \ldots, X_r^{(0)})$, the "systematic" Gibbs sampler in Algorithm 1 generates a sequence $X^{(0)}, \ldots, X^{(k)}$ of random vectors. The function GENCOND(x_{-i}, i) invoked in Step 12 generates a sample from $h_i^*(\cdot\,|x_{-i})$. Since each new sample is generated recursively from the previous sample, the sequence $\{X^{(j)}\}$ forms a Markov chain. The Gibbs sampler is a special case of a *Markov Chain Monte Carlo (MCMC)* sampling technique. In our database setting, the Gibbs sampler is more appealing that other MCMC methods because it is extremely generic in nature, requiring no special knowledge about the distribution h. Crucially, if the initial sample $X^{(0)}$ is generated from h, then the chain will be *stationary* in that every subsequent sample will be distributed according to h [1, Th. XIII.5.1]. Although the samples are not statistically independent, under mild regularity conditions the random vectors $X^{(0)}$ and $X^{(k)}$ become increasingly independent as k increases.[2] This "convergence to independence" is usually exponentially fast, so that k need not be very large. (In our experiments taking $k = 1$ sufficed.) Thus, if two chains $\{X^{(j)}\}$ and $\{Y^{(j)}\}$ start from the same state $X^{(0)} = Y^{(0)}$ but the respective Gibbs updates are performed independently for the chains, then $X^{(k)}$ and $Y^{(k)}$ will become approximately independent as k increases, again typically at an exponentially fast rate.

Algorithm 2 Rejection algorithm for example Gibbs sampler

```
1: GENCOND($v, i$):
2: repeat
3:     Generate $u$ according to $h_i$
4: until $Q(u \oplus_i v) \geq c$
5: Return $u$
```

We illustrate the method further with an example that is pertinent to our tail-sampling problem. Suppose that r is large and that the components of X are statistically independent, so that $h(x) = \prod_{i=1}^r h_i(x_i)$, where $h_i(u) = P(X_i = u)$ is the marginal distribution of X_i for $1 \leq i \leq r$. We assume that it is straightforward to generate samples from each h_i. Let Q be a real-valued function defined on $\mathcal{X}^r$, and suppose that for some value c we want to generate samples from the conditional distribution

$$h(x; c) = P(X = x \mid Q(X) \geq c) = h(x)I\big(Q(x) \geq c\big)/p_c.$$

Here $I(A) = 1$ if event A occurs and $I(A) = 0$ otherwise, and $p_c = P(Q(X) \geq c) = \sum_{x \in \mathcal{X}^r : Q(x) \geq c} h(x)$. If the cardinality of $\mathcal{X}$ is very large, then p_c is extremely hard to compute, so that $h(x; c)$ is known only up to the constant of proportionality p_c, and direct generation of samples from $h(x; c)$ is nontrivial.

We can, however, apply the Gibbs sampler. Note that, by the independence of the components of X, we have $h_i^*(u|v) = P\big(X_i = u \mid Q(X_i \oplus_i v) \geq c\big)$. Here $u \oplus_i v$ denotes the vector $(v_1, \ldots, v_{i-1}, u, v_i, \ldots, v_{k-1})$. Thus, samples from h_i^* in Step 12 can be generated by a rejection algorithm (Algorithm 2). To illustrate one especially simple case, suppose that $Q(x) = x_1 + \cdots + x_r$. Then $Q(u \oplus_i v) = u + \sum_j v_j$, so that $h_i^*(u|v) = P(X_i = u \mid X_i \geq c - \sum_j v_j)$. An efficient implementation of Algorithm 2 would operate on an existing Gibbs iterate $x \in \mathcal{X}^r$ by subtracting x_i from $Q(x)$, thereby computing $q^- = \sum_{j \neq i} x_j$, and then generating samples u from h_i until $u + q^- \geq c$. In the following, we write GIBBS$(X^{(0)}, k, c)$ and GENCOND(v, i, c) to indicate the dependence of these two algorithms on the parameter c.

To see the relevance of the above example to MCDB-R, consider a database consisting of a single table R having a single, random attribute A and exactly r tuples in all possible worlds (e.g., like the `Losses` table in Section 2 with the `CID` attribute dropped). Suppose that we interpret X_i above as the random variable corresponding to the value of the ith tuple ($1 \leq i \leq r$) and Q as an aggregation query over the values in R. Then the foregoing discussion shows that if we have somehow obtained a DB instance $D^{(0)}$ sampled from $h(x; c)$—i.e., an instance that is "large" in the sense that $Q(D^{(0)}) \geq c$—we can run the Gibbs sampler with the rejection algorithm for k steps to obtain another random instance $D^{(k)}$ that is (approximately) independent of $D^{(0)}$ and also 'large"—i.e., also distributed according to $h(x; c)$. That is, once we have obtained a DB instance corresponding to an upper tail of the query-result

[2]More precisely, the Markov chain associated with the Gibbs sampler is usually "irreducible" and "aperiodic" in an appropriate sense, and the chain "mixes" at an exponential rate; see [3].

distribution, we can generate additional, independent instances that also yield query results lying in the tail.

In the general MCDB-R setting, we can still interpret X as a random vector in which each component corresponds to an uncertain value in the database, but the components of X decompose into mutually independent blocks, where the variables within a block are dependent and are all generated via a call to a specified VG function. Moreover, the number of variables in a block—i.e., the number of values returned by a VG function—may vary over DB instances. Although formal discussion of Gibbs sampling at this level of generality would lead to very cumbersome notation, the basic idea is the same as above.

3.2 Rare-Event Simulation

The foregoing discussion shows that Gibbs sampling can be used to generate essentially independent samples from a tail of the query-answer distribution, starting from a "large" DB instance. However, such a starter instance can be hard to obtain, even when the quantile that defines the tail is known a priori. To address this challenging problem, we adapt ideas from the literature on "rare-event simulation"—see, e.g., [17] for an overview. The primary goal of rare-event simulation is to estimate the probabilities of very infrequent events; quantile estimation has received very little attention.

The two main techniques for rare-event simulation are *importance sampling* (IS) and *cloning*, also called *splitting*. The idea in IS is to sample from a modified possible-DB distribution (by suitably modifying the VG functions) such that extreme DB instances are generated with high frequency. Due to numerical instability, however, IS is known to behave badly for high dimensional problems [2]. In our setting, the effective dimension typically is roughly equal to the number of tuples in a relation, rendering IS unusable. We therefore look to cloning techniques.

The cloning approach has been of interest to the simulation community for a long time, but only recent work [2, 4, 18] has considered the type of "static" Monte Carlo simulations that are of interest in our setting. These algorithms start with a set of n randomly generated "particles." Each particle gives rise to a Markov chain that is obtained by repeatedly applying a random MCMC update ("perturbation") to the particle. The set of particles is "enriched" by deleting "non-extreme" particles and cloning "extreme" particles. The "Gibbs cloning" framework of Rubinstein [18] provides a general formulation of these ideas that we adapt for our purposes.

3.3 Adapting the Gibbs Cloner

Adapting and specializing the Gibbs cloning framework to our setting—the application to quantiles is novel—yields the basic procedure of Algorithm 3. In the algorithm, we represent a random database D as a vector of random variables, as per our prior discussion. (We treat each deterministic data value c as a random variable that is equal to c with probability 1.) In the algorithm, we maintain a set $\mathcal{S}$ of DB instances over a sequence of *bootstrapping* steps, so called because we are "bootstrapping" our way out to the tail. At the start of the ith step, $\mathcal{S}$ contains n_i instances, and the algorithm proceeds by (1) purging all but the top $100p_i\%$ "elite" (most extreme) instances, (2) cloning the elite instances to increase the size of $\mathcal{S}$ up to n_{i+1}, and then (3) perturbing each instance in $\mathcal{S}$ using the Gibbs sampler, in order to re-establish (approximate) independence while ensuring that all instances yield query results that lie in the current tail. The function $\text{CLONE}(\mathcal{S}, n)$ operates by simply duplicating each DB instance in $\mathcal{S}$ approximately $n/|\mathcal{S}|$ times.

The Appendix describes how to choose the algorithm parameters to efficiently control the relative difference between the desired upper-tail probability p and the actual probability of the upper tail

Algorithm 3 Basic tail-sampling algorithm

```
1: Inputs:
2:     $p$: target upper-tail probability
3:     $l$: desired number of tail samples
4: Outputs:
5:     $\hat{\gamma}$: estimate of $(1-p)$-quantile
6:     $\mathcal{S}$: set of $l$ samples from $h(\,\cdot\,;\hat{\gamma})$
7: Parameters
8:     $k$: number of Gibbs updating steps
9:     $m$: number of bootstrapping steps
10:    $n_1, n_2, \ldots, n_m$: intermediate sample sizes
11:    $p_1, p_2, \ldots, p_m$: intermediate tail probabilities
12:
13: // Initialize
14: Generate databases $D^{(1)}, \ldots, D^{(n_1)}$ i.i.d. according to $h(\,\cdot\,)$
15: $\mathcal{S} \leftarrow \{\, D^{(1)}, D^{(2)}, \ldots, D^{(n_1)} \,\}$
16: $n_{m+1} \leftarrow l$
17: // Execute $m$ bootstrapping steps
18: for $i \leftarrow 1$ to $m$ do
19:     $\hat{\gamma}_i \leftarrow$ the $(p_i|\mathcal{S}|)$-largest element of $\{\, Q(D) : D \in \mathcal{S} \,\}$
20:     Discard all elements $D \in \mathcal{S}$ with $Q(D) < \hat{\gamma}_i$
21:     $\mathcal{S} \leftarrow \text{CLONE}(\mathcal{S}, n_{i+1})$
22:     for $D \in \mathcal{S}$ do                          // Gibbs-update step
23:         $D \leftarrow \text{GIBBS}(D, k, \hat{\gamma}_i)$    // GIBBS defined as in Sec. 3.1
24:     end for
25: end for
26: return $\hat{\gamma} = \hat{\gamma}_m$ and $\mathcal{S}$
```

defined by the quantile estimator $\hat{\gamma}$. In particular, we show that this relative error is minimized by setting $n_i = n$ and $p_i = p^{1/m}$ for $1 \leq i \leq m$. Here $n = N/m$, where N is the total number of samples over all bootstrapping steps; increasing N increases cost but improves accuracy. We assume these values throughout.

After the ith purge, the smallest query-result value for the retained elite samples is an estimate of γ_i, defined to be the $(1 - p^{i/m})$-quantile of the query-result distribution; thus the γ_i's are increasing and γ_m corresponds to the extreme quantile of interest. At each step i, we estimate a $1-p^{1/m}$ quantile of F_{i-1}, the conditional distribution of the query result $Q(D)$, given that $Q(D) \geq \gamma_{i-1}$. (Here we take $\gamma_0 = -\infty$.) For typical values of, say, $p = 0.001$ and $m = 4$, we see that even though we ultimately need to estimate an extreme 0.999-quantile, at each step we merely need to estimate a 0.82-quantile, a much easier problem.

4. IMPLEMENTATION OVERVIEW

In the following sections, we describe how we implemented tail sampling (or *TS* for short) as in Algorithm 3. The goal is to preserve, to the extent possible, the computational efficiency of the MCDB tuple-bundle processing mechanism as discussed previously.

4.1 The Gibbs Looper and Data Streams

In the MCDB-R setting, we call the iterative process in Algorithm 3 the *Gibbs Looper*. There are two key variables that track the progress of the this algorithm: `cutoff` and `curQuantile`. In terms of our previous notation, `cutoff` is the γ_i value that defines the current tail and, after i iterations, `curQuantile` $= p^{i/m}$ is the probability that a query-result instance falls in the current tail.

At each iteration of the Gibbs Looper, each of the n versions (i.e., realized instances) of the database is randomly perturbed by the Gibbs sampler to form n new DB versions. Initially, a cloned database has the same contents as its parent, but as each DB version is perturbed by its Gibbs sampler, parent and child will drift apart. To "fuel" the perturbation, the Gibbs sampler needs access to a stream of random data. There is a data stream associated with every uncertain data value (or correlated set of uncertain data values) in the database. E.g., consider the random `Losses` table from Section 2. The tuple bundle corresponding to a given customer is initially enhanced with a seed for use by the pseudorandom number generator (PRNG). Repeated execution of the `Normal` VG function, parameterized with the customer's mean loss value `m`, produces a stream of realized loss values for the customer. In MCDB, the first n elements of the stream simply correspond to the customer's losses in n Monte Carlo repetitions, i.e., in n generated DB instances. In MCDB-R, because of the rejection algorithm (Algorithm 2) used in the Gibbs sampler, there is a more complicated mapping from DB instance to position in the stream, but the Gibbs sampler nonetheless goes to the stream whenever it needs a loss value for the customer. We will often refer to a stream of data via the PRNG seed that was used to produce it.

In general, a VG function may produce a table of correlated data values when called, so an "element" of a stream may actually comprise a set of values. Also, a given PRNG seed may occur in multiple tuples—due to a 1-to-m join of the random table to another table—or multiple times in a tuple bundle due to a self-join.

4.2 An Example

We illustrate the Gibbs Looper in more detail using the total-loss query of Section 2 as an example; for brevity, we drop the `CID` column from the `Losses` and `means` tables, as well as the selection predicate on `CID` from the query itself. Suppose that there are three customers and that the corresponding three rows in the `means` table have values 3.0, 4.0, and 5.0, respectively. Also suppose that $p = 1/32$ and $n = 4$, so that we want to generate a sample of four total losses that are each in the top 3.125% of the total-loss (i.e., query-result) distribution. We will use $m = 5$ iterations of the Gibbs Looper to generate the samples.

First, the underlying VG function is used to generate three streams of random data, as depicted in Fig. 1. The first four values produced by each VG function are assigned, in sequence, to each of the $n = 4$ DB versions, as in Fig. 1(a). After this initialization, iteration $i = 1$ of the Gibbs Looper begins. First, the two DB versions with final `SUM` values in the lowest $100(1 - p^{1/m})\%$ percentile (50% in our example) are discarded, and the remaining two elite DB versions are *cloned*. "Cloning" is accomplished by duplicating the assignment of random values from each stream of random data to the various DB versions, as depicted in Fig. 1(b). The current value for `cutoff` at this point is set to be 12.07, since this is the aggregate value associated with the least-extreme remaining DB version. `cutoff` also serves as an estimate for the point at which there is only a $100p^{1/m}\%$ ($= 50\%$) chance of seeing a larger value for the answer to the `SUM` query.

Since two of the DB versions have been duplicated, it is necessary to perturb all of the versions so that they are sufficiently differentiated. In our example, the Gibbs sampler begins the perturbation by attempting to replace the value 3.26 that is associated with the first stream of loss values (the stream associated with mean value `3.0`). This is done by considering the next unassigned random value associated with mean `3.0`—indicated in bold in Fig. 1(b). Unfortunately, this value (which is 3.24) cannot be used to replace 3.26, because it would result in an overall aggregate value of 12.05, which does not meet or exceed the current value of `cutoff` = 12.07. Thus, the 3.24 is rejected. Next, the 5.13 is considered. Since the 5.13 would result in an overall aggregate value of 13.94 (which exceeds `cutoff`), the 5.13 is accepted.

The procedure is repeated using the other two streams. The 4.59 associated with `mean.val` = 4.0 is replaced with 4.55 (acceptable since this decreases the aggregate value to 13.90, which exceeds the cutoff of 12.07), and the 4.22 associated with `mean.val` = 5.0 is replaced with 4.29, resulting in the state of Fig. 1(c).

DB version two is updated in the same way. The value 3.26 is replaced with 4.07 (acceptable, since the new aggregate value is 12.88). An attempt is made to replace the 4.59 associated with `mean.val` = 4.0 using 3.68. However, this would reduce the overall aggregate for DB version two to 11.97, which does not meet or exceed the current value for `cutoff`. So the 3.68 is rejected, and 5.16 (the next unassigned value associated with `mean.val` = 4.0) is chosen. Finally, the 4.22 is replaced with 5.77, resulting in Fig. 1(d). After DB versions three and four are updated, the result is Fig. 1(e). Then iteration $i = 2$ begins, and cloning the top two DB versions gives us Fig. 1(f). This whole process is repeated until the end of iteration $i = m$.

4.3 Implementing A Gibbs Looper

In practice, there are many possible ways to implement a Gibbs Looper in the presence of complicated database operations such as joins. The simplest would maintain n different random versions of the database. Then, in each iteration of the Gibbs looper (where an "iteration" is the process of perturbing all n DB versions) and for each DB version, we would undertake the following steps. For each stream of random data, we would (1) replace the current stream element that is being used in the DB version with a new stream element, (2) run the query over the modified DB version, (3) accept the new stream element if the new query result exceeds `cutoff` and otherwise try steps (1) and (2) again.

The drawback of such an implementation is that it requires running the underlying query many, many times over many, many random databases. For example, imagine that we will maintain $n = 100$ DB versions, each requiring a modest one million streams of random data. Imagine that the Gibbs Looper needs ten iterations to complete and that we need to reject (on average) ten random values before one is accepted. In this case, we will need to run an entire query plan $100 \times 10^6 \times 10 \times 10 = 10^{10}$ different times to complete the TS procedure, which is obviously unacceptable.

We therefore use MCDB's tuple-bundle trick of running the underlying query only once, no matter how many DB versions are needed. The extensions to MCDB's tuple-bundle mechanism are many and intricate; we now sketch the key ideas.

5. GIBBS TUPLE BUNDLES

As with MCDB, query execution in MCDB-R is organized around the idea of bundling together the different versions of a tuple over the generated DB instances. To execute a query, MCDB-R runs a plan that pushes a stream of instantiated tuple bundles into a special `GibbsLooper` operation. These bundles must have all of the information necessary to implement the process illustrated in Fig. 1, i.e., given a DB version and a stream of random data, the Gibbs Looper must be able to determine which random value from the stream is assigned to the DB version, and then efficiently determine the effect of an update to the value on the final query result. This requires that MCDB tuple bundles must be augmented with a notion of "lineage" that links each random value in the database to the stream from which it came.

Gibbs tuples therefore differ from MCDB tuple bundles. Instead of containing a PRNG seed, a Gibbs tuple contains a pointer to a "tail-sampling seed", or TS-seed for short. This object, described in Section 6 below, augments the basic PRNG seed with "bookkeeping" information needed by the Gibbs Looper. In an MCDB bun-

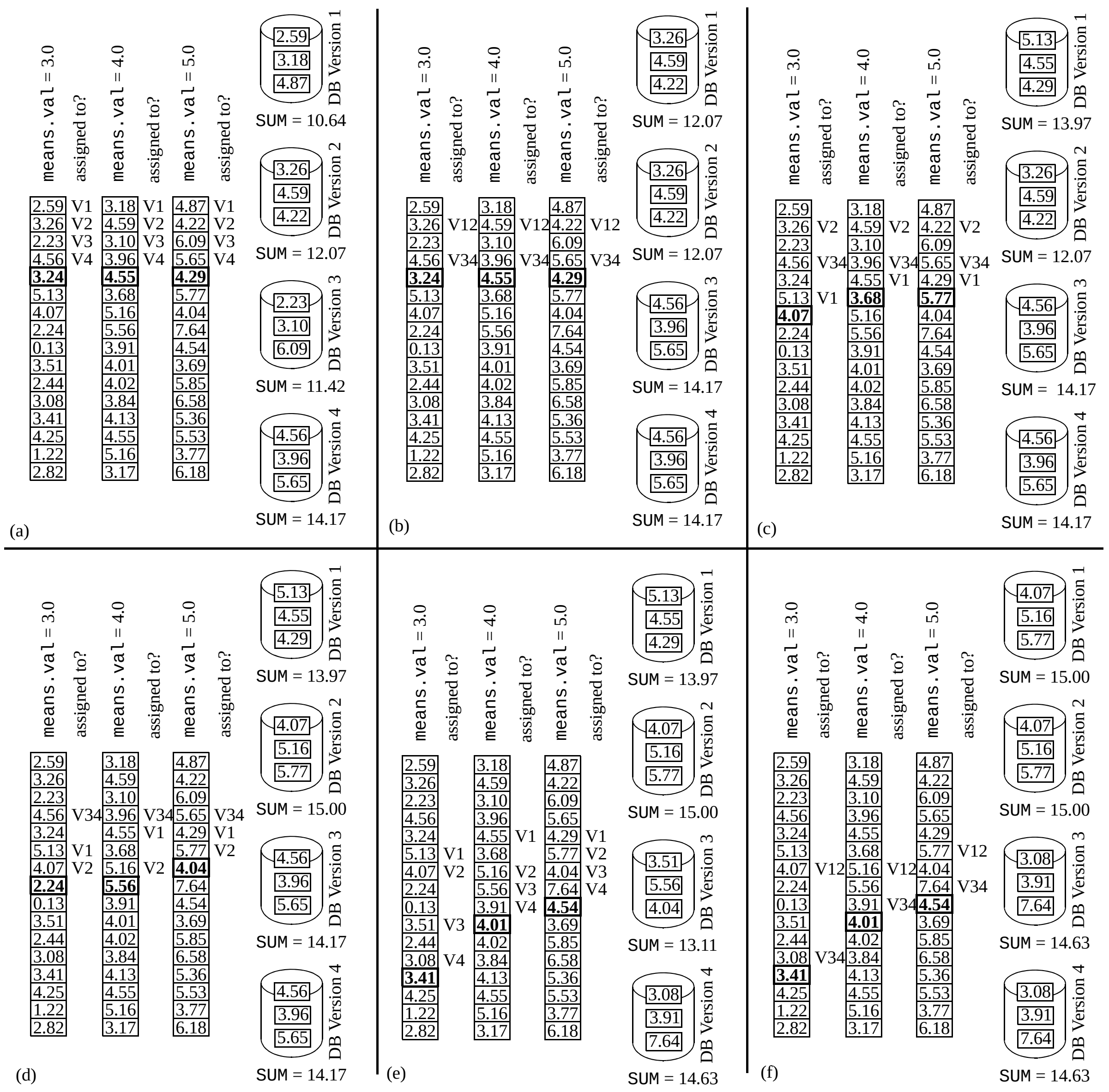

Figure 1: Running the tail-sampling process to obtain four DB instances

dle, PRNG seeds may be discarded if they are no longer needed. In MCDB-R, a TS-seed handle can never be discarded, because each random value must always point to its associated PRNG (i.e., associated stream). Another important difference is that, whereas an instantiated MCDB tuple bundle contains exactly n stream elements per random value—one for each Monte Carlo repetition—an instantiated Gibbs bundle contains many more elements, because the Gibbs Looper can use up many stream elements during the Gibbs perturbation process.[3] Indeed, the Gibbs Looper may run out of data during processing and need to re-execute part of the query plan (see Section 9). The number of stream elements to instantiate in a Gibbs tuple is chosen to trade off the cost of carrying lots of data throughout the query plan with the cost of re-executing part of the query if the data runs out.

For an example of how Gibbs tuples are pushed through a query plan, consider the following query, which determines a company's total salary "inversion," resulting from certain employees who earn more than their managers:

```
SELECT SUM(emp2.sal - emp1.sal)
FROM emp AS emp1, emp AS emp2, sup
WHERE sup.boss = emp1.eid AND emp1.sal < 90K
  AND sup.peon =  emp2.eid AND emp2.sal > 25K
  AND emp2.sal > emp1.sal
```

Imagine that the attribute `emp.sal` is random, and we want three samples from the tail of the query-result distribution. The resulting query plan and execution of this query is illustrated in Fig. 2.

This plan operates over Gibbs tuples rather than ordinary tuples. An instantiated Gibbs tuple can have one or more arrays of random values; every array of random values is associated with exactly one PRNG seed. A Gibbs tuple can also have one or more *isPres* attributes. An array of *isPres* values is created when a selection predicate is applied to a random attribute, and indicates for each DB instance whether or not the predicate is satisfied. If the predicate

[3]Because instantiated values are often needed to prepare tuple bundles for the Gibbs Looper—see the example below—they cannot be simply generated on the fly during Gibbs-Looper operation.

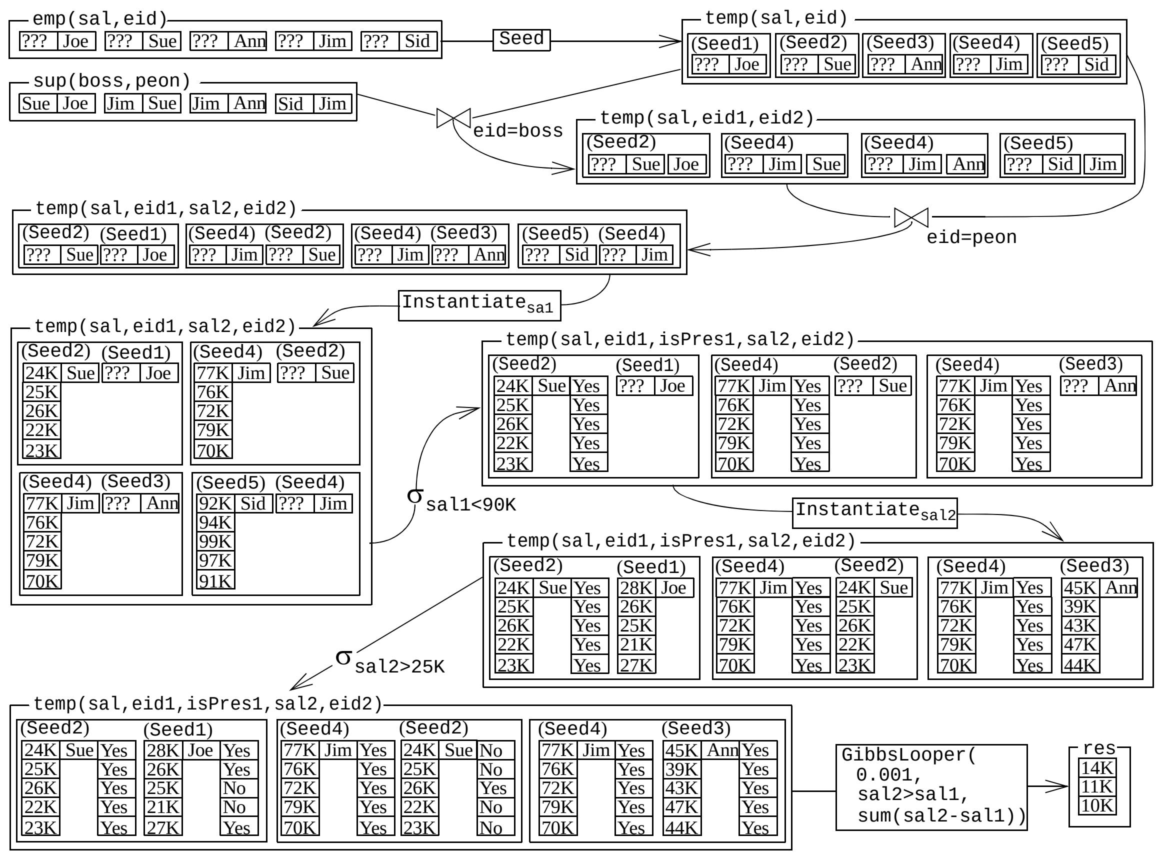

Figure 2: Gibbs-tuple processing over a multi-table query plan

is not satisfied in any DB instance, then the entire Gibbs tuple is dropped. (Unlike MCDB, we cannot simply record *isPres* values for the entire tuple, since attribute values change individually during Gibbs sampling.) Finally, note that the query plan makes use of the MCDB-R `Seed` and `Instantiate` operations. The former operation attaches the handle for a TS-seed to each Gibbs tuple, and the latter operation uses a PRNG seed to attach a subsequence from a stream of random values to the Gibbs tuple.

6. TAIL-SAMPLING SEEDS

A TS-seed contains both the PRNG seed needed to instantiate a stream of random values and the information needed to map the current value of an attribute in a DB version to a position in the stream. This mapping is needed to compute a query-result value for the current DB version. E.g., in Fig. 2, the query plan produces a set of three Gibbs tuples that are piped into the `GibbsLooper` operation. Suppose that we have the assignment (`seed1`, 1), (`seed2`, 1), (`seed3`, 2), (`seed4`, 2) for a DB version. That is, Joe's salary is \$28K (the first value in his stream), Ann's salary is \$39K (the second value in her stream), and so forth. Then the query result for this DB version is (\$28K - \$24K) = \$4K, since Joe makes \$4K more than Sue and no other employee makes more than their boss.

In MCDB-R, a TS-seed contains (1) a TS-seed identifier, (2) the actual PRNG seed used to produce a stream of random data, (3) the range of stream values currently materialized and present within the Gibbs tuples, (4) the last random value in that range that has previously been assigned to any DB version for this TS-seed, and (5) the random value currently assigned to each DB version for this TS-seed. Item (5) is the mapping described above, and item (4) is needed during Gibbs rejection sampling to obtain a new random value. Item (3) is needed to quickly determine whether the Gibbs Looper has run out of data and, if so, the position in the stream from which to start replenishing. A call to the `Seed` operator both adds a seed handle to a Gibbs tuple and creates the actual TS-seed data structure for the tuple, which is written to disk.

7. THE GIBBS LOOPER

The Appendix describes the `GibbsLooper` operator in detail. A key feature is that it switches the inner and outer **for** loops of Algorithm 3. Rather than perturb DB versions one at a time, looping through the random data values (i.e., the TS-seeds), the algorithm perturbs data values one at a time, looping through the DB versions, thereby amortizing expensive data scans.

Specifically, `GibbsLooper` inserts the Gibbs tuples into a disk-based priority queue; the initial sort key for a Gibbs tuple is the smallest TS-seed handle in the tuple. `GibbsLooper` then iterates through the overall set of TS-seed handles in increasing order. For each TS-seed handle, it loads the TS-seed—and all "associated" Gibbs tuples that contain at least one handle for the TS-seed—into memory. Then, for the first DB version, it consults the TS-seed to determine the first unused stream value to try in the rejection algorithm. It uses the associated Gibbs tuples to compute the changed value of the query result for the DB version due to the new stream value, and then checks if the new query-result value still exceeds `cutoff`. If so, then `GibbsLooper` accepts the new stream value and moves on to update the next DB version. If not, it tries again until an appropriate stream value is found. Once all DB versions have been updated, it reinserts all of the Gibbs tuples into the priority queue—first updating the sort key for each reinserted tuple to be the next largest TS-seed handle in the tuple—and moves on to the next overall TS-seed handle. (Thus a Gibbs tuple will be processed multiple times if it contains multiple TS-seed handles.)

`GibbsLooper` is efficient because it essentially merges Gibbs tuples in the disk-based priority queue with a sorted file containing all of the TS-seeds. When a TS-seed is loaded into memory, the priority queue is used to quickly obtain all of the Gibbs tuples that could possibly be affected by an update of the TS-seed.

8. JOINS ON RANDOM ATTRIBUTES

One issue that demands more explanation is how `GibbsLooper` handles join predicates on random attributes. E.g., suppose R_1 joins with R_2 on a random attribute, and R_2 joins with R_3 on a deterministic attribute. Some t_2 in R_2 is modified when one of its random attributes is updated, so it joins with both a t_1 from R_1 and a t_3 from R_3—the result is that a new tuple "pops" into existence.

Fortunately, this case is easily handled by MCDB-R. When the underlying query plan is run (before `GibbsLooper` is ever invoked), original MCDB's `Split` operation is run on any random attribute or attributes that are to be joined. This operation changes the attributes so that they are no longer random and transfers any non-determinism to the *isPres* attribute. E.g., consider a Gibbs tuple g with schema `(fname, age)` where attribute `fname` is constant, and attribute `age` is non-constant. Specifically, `age = 20` for the first and third values in the stream of random data stored in the Gibbs tuple, and `age = 21` for the second and fourth values: $g =$ `(Jane,(20,21,20,21),(Y,Y,Y,Y))`, where the last nested vector contains the *isPres* values, and indicates that Jane appeared in all four Monte Carlo repetitions (though with varying ages). An application of the `Split` operation to t with $Atts = \{\texttt{age}\}$ yields two Gibbs tuples: $g_1 =$ `(Jane,20,(Y,N,Y,N))` and $g_2 =$ `(Jane,21,(N,Y,N,Y))`. Then, any join on the `age` attribute is a join on a deterministic attribute, and we will not have the problem of new Gibbs tuples "popping" into existence, since two sets of Gibbs tuples (each associated with one of `Jane`'s possible ages) would both be created and sent to `GibbsLooper`. Furthermore, only one of them can be valid at a time: if, e.g., `Jane`'s age is updated to be `21`, then the mapping information within the TS-seed associated with `Jane`'s age is updated as well, causing any Gibbs tuple associated with an age of `20` to become invalid.

9. RE-RUNNING THE QUERY PLAN

Since each Gibbs tuple contains only a finite amount of data, the Gibbs Looper may run out of data while perturbing a DB version. If so, MCDB-R discards all existing Gibbs tuples, empties the `GibbsLooper` priority queue, and then runs the entire query plan again to produce the required data. The newly-created Gibbs tuples that contain the data are piped into the `GibbsLooper` operation.

During such a "replenishing" run, query-plan execution differs from the first time around in a couple of ways. First, the result of each deterministic part of the query plan is materialized and saved during the first run, avoiding the need to repeat these computations during replenishment. Second, the `Instantiate` operation never adds stream values to a Gibbs tuple that have already been processed; it only adds new or currently assigned values.

10. CONCLUSION

MCDB-R extends the MCDB system to permit risk evaluation in the database. Space constraints have forced us to exclude some important material from the paper. Some very brief benchmarking results, given in the Appendix, provide evidence for the accuracy of the method and show a reduction in processing time from roughly 18 hours in MCDB to 11 minutes in MCDB-R.

Acknowledgements This work was supported in part by the US DoE under grant no. DE-SC0001779 and the US NSF under grant no. 0915315, and by a Sloan Foundation Fellowship.

11. REFERENCES

[1] S. Asmussen and P. W. Glynn. *Stochastic Simulation: Algorithms and Analysis*. Springer, 2007.

[2] Z. I. Botev and D. P. Kroese. An efficient algorithm for rare-event probability estimation, combinatorial optimization, and counting. *Methodol. Comput. Appl. Prob.*, 10:471–505, 2008.

[3] R. C. Bradley. Basic properties of strong mixing conditions: A survey and some open questions. *Probab. Surveys*, 2:107–144, 2005.

[4] F. Cérou, P. D. Moral, T. Furon, and A. Guyader. Rare event simulation for a static distribution. INRIA Research Report 6792, Rennes, France, 2009.

[5] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

[6] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009.

[7] N. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

[8] A. Das Sarma, O. Benjelloun, A. Y. Halevy, S. U. Nabar, and J. Widom. Representing uncertain data: models, properties, and algorithms. *VLDB J.*, 18(5):989–1019, 2009.

[9] H. A. David and H. N. Nagaraja. *Order Statistics*. Wiley, third edition, 2003.

[10] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD*, pages 73–84, 2006.

[11] S. Geman and D. Geman. Stochastic relaxation, Gibbs distribution and the Bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intelligence*, 6(6):721–741, 1984.

[12] S. Guha. RHIPE - R and Hadoop Integrated Processing Environment. `http://ml.stat.purdue.edu/rhipe/`.

[13] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. MCDB: a Monte Carlo approach to managing uncertain data. In *ACM SIGMOD*, pages 687–700, 2008.

[14] O. Kennedy and C. Koch. PIP: A database system for great and small expectations. In *ICDE*, pages 157–168, 2010.

[15] C. Koch and D. Olteanu. Conditioning probabilistic databases. *PVLDB*, 1(1):313–325, 2008.

[16] A. J. McNeil, R. Frey, and P. Embrechts. *Quantitative Risk Management: Concepts, Techniques, and Tools*. Princeton University Press, 2005.

[17] G. Rubino and B. Tuffin, editors. *Rare Event Simulation Using Monte Carlo*. Wiley, 2009.

[18] R. Rubinstein. The Gibbs cloner for combinatorial optimization, counting, and sampling. *Methodol. Comput. Appl. Prob.*, 11(4):491–549, 2009.

[19] R. J. Serfling. *Approximation Theorems of Mathematical Statistics*. Wiley, 1980.

[20] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and SciDB. In *CIDR*, page 26, 2009.

[21] A. Thiagarajan and S. Madden. Querying continuous functions in a database system. In *SIGMOD*, pages 791–804, 2008.

APPENDIX

A. THE GIBBS LOOPER IN DETAIL

A.1 Initialization

Aside from n, m, and p (see Section 3.3), `GibbsLooper` takes as input five principal arguments:

1. The pipeline producing the input stream of Gibbs tuples.
2. The aggregate operation from whose tail distribution `GibbsLooper` will sample, e.g., `sum(sal2-sal1)` in Fig. 2.
3. The final selection predicate (e.g., `sal2 > sal1` in Fig. 2) that must be applied to each tuple prior to inclusion of the tuple's value in the aggregate. In particular, any selection predicate that references random attributes generated by more than one PRNG seed must be pulled up into `GibbsLooper`. Such a predicate cannot be applied lower in the query plan, because the Gibbs sampling will arbitrarily combine random values from the streams associated with different PRNG seeds, and it is not possible to know the outcome of the predicate before `GibbsLooper` runs its Gibbs sampler.
4. The grouping attributes (if any) for the query.[4]
5. A file containing all of the TS-seed objects, sorted on each object's handle.

Given these inputs, `GibbsLooper` initially processes the stream of incoming Gibbs tuples by computing the initial aggregate value for each of the n DB versions and inserting the tuples into a disk-based priority queue, sorted by the smallest TS-seed handle in each Gibbs tuple (as discussed in Section 7). The initial aggregate values are computed for the TS-seed mapping in which the ith value in each stream is mapped to the ith DB version for $1 \leq i \leq n$. Next, `GibbsLooper` discards the "non-elite" DB versions whose aggregate totals are in the bottom $100(1 - p^{1/m})\%$ by overwriting them with clones of the elite versions. The `cutoff` value is set equal to the smallest aggregate value. The overwriting of a non-elite DB version by a clone of an elite version is accomplished during a single read/write pass through the TS-seed file (see below).

A.2 Performing the Perturbation

At this point, the `GibbsLooper` is ready to begin perturbing the various DB versions using a Gibbs sampler. The manner in which the perturbation is accomplished is best illustrated using an example, given as Fig. 3.

Fig. 3(a) begins with the three Gibbs tuples that were output from the query plan of Fig. 2, inserted into the `GibbsLooper`'s priority queue. (For simplicity, we drop the *isPres* attributes and look at only two DB instances.) The four TS-seeds are also depicted, as are the two DB versions that correspond to the initial TS-seed mapping (described above). The goal at this point to perturb these versions via Gibbs sampling, subject to the constraint that the total aggregate value must meet or exceed `cutoff` = \$1K. Note that while the evolving DB versions are depicted in the figure, they are never actually materialized; they are completely determined, however, by the current state of the Gibbs tuples and the TS-seeds.

To obtain two new DB versions with `SUM` values that meet or exceed `cutoff`, the first TS-seed handle, (`seed1`), is considered, and the lone associated Gibbs tuple is removed from the priority queue, as in Fig. 3(b). DB version one is updated by assigning the first unassigned random value associated with `seed1` to the DB version, transforming (24K, Sue, 28K, Joe) into (24K, Sue, 25K, Joe). This decreases the final aggregate result from \$4 to \$1K, since Joe's salary exceeds Sue's by \$1K. Because the total salary inversion still meets the `cutoff` of \$1K, the update is accepted.

Next, we attempt to update DB version two. The next unassigned random value in the stream associated with `seed1` is the fourth one, which transforms (25K, Sue, 26K, Joe) into (25K, Sue, 21K, Joe). This changes the final aggregate value to \$0, and so this proposed update is rejected. Next we try the fifth random value in the stream. This gives us (25K, Sue, 27K, Joe), which increases the final aggregate value to \$2K, and so it is accepted. We are now done with `seed1`, and so we reinsert the one Gibbs tuple associated with this seed back into the priority queue, after first updating its sort key to its next largest seed handle, (`seed2`); see Fig. 3(c).

As shown in Fig. 3(d), the two Gibbs tuples associated with (`seed2`) are then removed from the priority queue. The assignments for `seed2`'s stream are updated in much the same way as for `seed1`; note, however, that `seed2` influences *two* tuples in each DB version. Once the update of `seed2` is complete and both of the associated Gibbs tuples have been reinserted back into the priority queue, we are left with Fig. 3(e). Note that the keys for these reinserted Gibbs tuples are now (`seed4`) and `infinity`. The value `infinity` is used because the random values associated with `seed1` and `seed2` have been each been updated—there is no reason to consider this Gibbs tuple again during this particular perturbation, so it is forced to the tail of the queue.

As illustrated in Fig. 3(f), we now move to (`seed3`). The two DB instances are updated with respect to `seed3`; because `seed3` control's Ann's salary and Ann has no impact on the query result, both updates are trivially accepted. The process continues with (`seed4`), which again has no effect on the final query result, at which point both DB versions are fully perturbed (Fig. 4(a)).

Since all of the TS-seed handles have been processed, it is now time to clone the elite DB versions and overwrite the non-elite versions. This process is illustrated by the transition from Fig. 4(a) to Fig. 4(b). In a single read/write pass, the column in each TS-seed that records the assignment for DB version two is simply copied to the column for version one. Now the first iteration of `GibbsLooper` has completed, and iteration two can begin.

B. WHEN WILL MCDB-R WORK BEST?

Algorithm 1—and hence Algorithm 3—will work best when, for each most-likely extreme database D, the query result $Q(D)$ is relatively insensitive to changes in the value of a single data value x. This scenario fails, e.g., for a `SUM` query in which data values $X_1, \ldots, X_r$ are i.i.d. according to a "subexponential" distribution such as lognormal or Pareto. Under such a heavy-tailed distribution, $P(X_1 + \cdots + X_k > c) \approx kP(X_1 > c)$ for large c; i.e., with high probability, the query result for an extreme database D—viewed as a vector y of values—will lie in the upper tail because some individual y_i is extremely large. After generating a new candidate value u for y_i in Algorithm 2, the value of $Q(u \oplus_i y)$ will likely drop sharply, and many candidates will be required prior to acceptance in Line 4. Our approach is thus unlikely to be universally applicable. That said, it is still useful for a wide class of risk-estimation problems, as evidenced by the extensive applied literature that pertains to extreme-value theory for light-tailed distributions and by the prevalence of aggregates, such as `SUM` and `AVG`, that have the insensitivity property under a light-tailed regime.

C. SETTING PARAMETERS

In this section, we address the key question of how to choose the parameters in Algorithm 3. Because the cost of a Gibbs-updating

[4]Grouping is handled by, in effect, treating a `GROUP BY` query over g groups as g separate, simultaneous queries, each with a selection predicate that limits the query to a specific group.

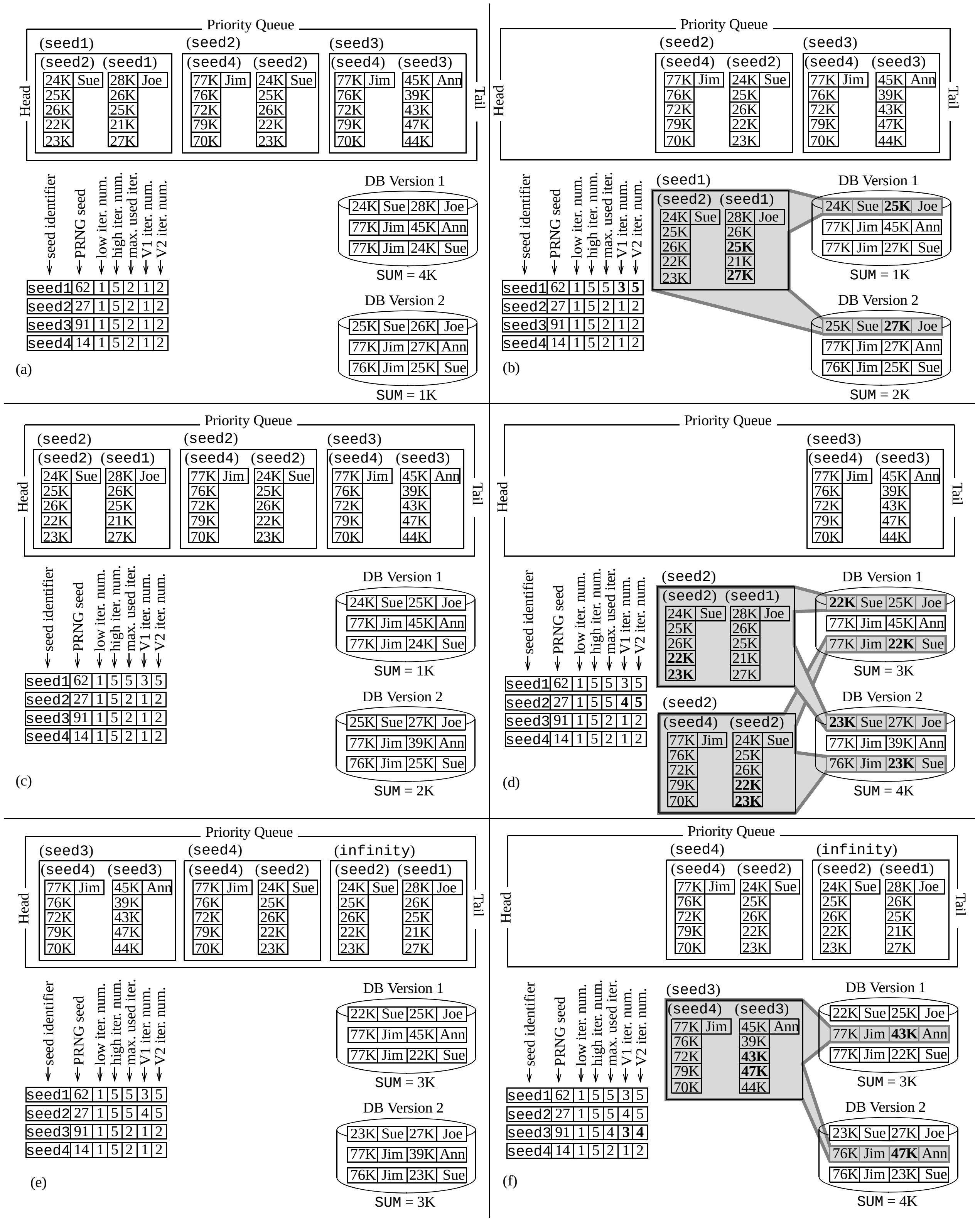
Figure 3: The perturbation process for the "salary inversion" query

iteration is proportional to the data size and hence extremely expensive, we choose the number of iterations k to be as small as possible, subject to the requirement that, after the Gibbs-update step in line 23, the n_{i+1} databases in $\mathcal{S}$ are approximately statistically independent. The MCMC literature provides a variety of procedures for determining k. Recall, however, that a sequence of Gibbs samples typically "forgets" its initial state exponentially fast, so that very small values of k suffice in practice; as mentioned previously,

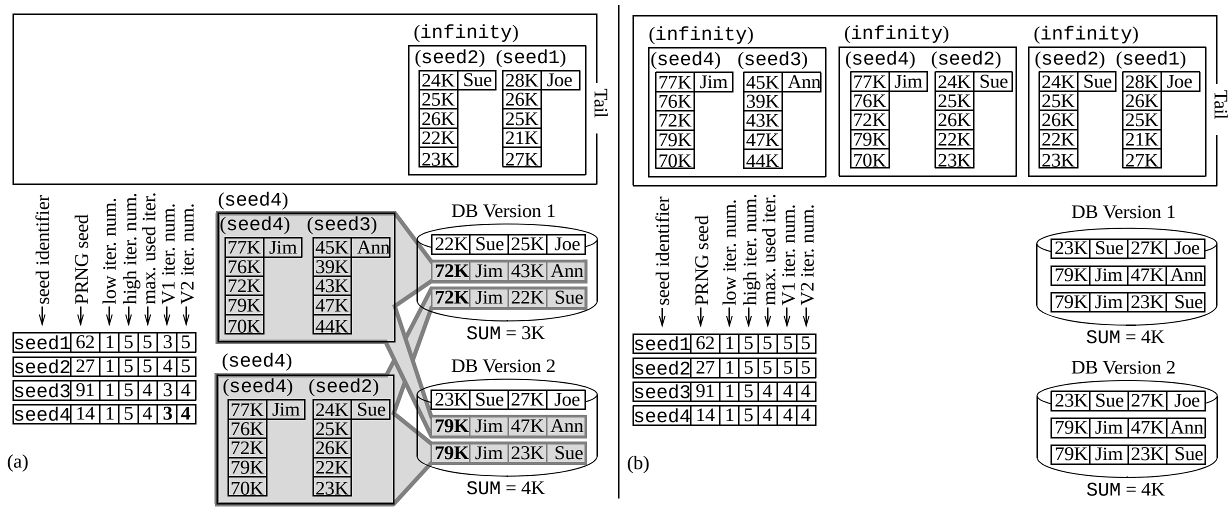

Figure 4: Perturbation example, continued

taking $k = 1$ worked well in experiments.

Now suppose that k has been fixed and is sufficiently large so that Gibbs updating yields samples that can be viewed as independent. Also suppose that the total number N of samples over all bootstrapping steps has been fixed. We consider the problem of selecting m, $n_1, \dots, n_m$ and $p_1, \dots, p_m$ to minimize statistical error.

One approach is to minimize the relative difference between the desired upper-tail probability p and the actual probability of the upper tail defined by the quantile estimator $\hat{\gamma}_m$. I.e., we solve the following optimization problem:

$$\underset{m, n_1, \dots, n_m, p_1, \dots, p_m}{\text{minimize}} \; E\Big[\Big(\frac{\bar{F}_0(\hat{\gamma}_m) - p}{p}\Big)^2\Big] \tag{1}$$

such that

$$\sum_{i=1}^{m} n_i = N \quad \text{and} \quad \prod_{i=1}^{m} p_i = p \tag{2}$$

$$p_i \in [0,1] \text{ and } n_i \in \{0, 1, \dots, N\}, \quad 1 \le i \le m \tag{3}$$

Here $F_0(x) = P\big(Q(D) < x\big)$ and $\bar{F}_0 = 1 - F_0$. Given a solution to the minimization problem (1)–(3), the overall parameter-selection task will reduce to selection of N. We refer to the minimand in (1) as the *mean-squared relative error* (MSRE).

Note that an alternative goal might be to control the relative error of $\hat{\gamma}_m$ directly. For light-tailed query-result distributions, which are the class of primary interest in this paper, minimizing MSRE rather than minimizing the relative error of $\hat{\gamma}_m$ typically leads to more stringent error control. E.g., suppose that, unknown to us, F_0 is the standard normal distribution, and that our desired tail probability is $p = 0.001$, so that $\gamma \approx 3.090$. If we allow the actual tail probability to deviate from p by no more than $\pm 1\%$, then the corresponding estimate of γ will lie approximately in the range $(3.087, 3.093)$, which represents a maximum error of about $\pm 0.1\%$.

In the following, we fix k and N as described above, and show how to select values of m, $n_1, \dots, n_m$, and $p_1, \dots, p_m$ that approximately solve the minimization problem (1)–(3). We assume that F_0 is continuous and strictly increasing.

We first abstract the process by which Algorithm 3 computes $\hat{\gamma}_m$. For real numbers $c_1, c_2, \dots, c_m$, set

$$F_1(x; c_1) = \frac{F_0(x) - F_0(c_1)}{1 - F_0(c_1)} = P\big(Q(D) < x \mid Q(D) \ge c_1\big),$$

and for $2 \le i \le m$ recursively define

$$F_i(x; c_1, \dots, c_i) = \frac{F_{i-1}(x; ; c_1, \dots, c_{i-1}) - F_{i-1}(c_i; c_1, \dots, c_{i-1})}{1 - F_{i-1}(c_i; c_1, \dots, c_{i-1})}$$

for any sequence $c_1, c_2, \dots, c_m$. An inductive argument establishes the identity $F_i(x; c_1, \dots, c_i) = F_1(x; c_i)$ for $2 \le i \le m$. For a set $X_1, X_2, \dots, X_n$ of i.i.d. samples from a strictly increasing continuous distribution function F, the *order statistics* are denoted as $X_{(1)} < X_{(2)} < \cdots < X_{(n)}$. Setting $r_i = n_i(1 - p_i)$ (rounded to the nearest integer), and assuming that k is sufficiently large as discussed previously, we can view Algorithm 3 as follows. The algorithm proceeds by generating n_1 i.i.d. samples $X_{0,1}, X_{0,2}, \dots, X_{0,n_1}$ from F_0. Then, for $i = 1, 2, \dots, m$, the algorithm (1) sets $\hat{\gamma}_i = X_{i-1,(r_i)}$ and (2) generates n_{i+1} samples $X_{i,1}, \dots, X_{i,n_{i+1}}$ from $F_i(\,\cdot\,; \hat{\gamma}_1, \dots, \hat{\gamma}_i) = F_1(\,\cdot\,; \hat{\gamma}_i)$.

To analyze the behavior of $\bar{F}_0(\hat{\gamma}_m)$, and hence the behavior of the MSRE, we employ the common tactic of reducing the analysis to the setting of uniform random variables. Specifically, let $\{U_{i,j} : 0 \le i \le m, 1 \le j \le n_{i+1}\}$ be a collection of i.i.d. random variables uniformly distributed on $[0,1]$. Consider a procedure that sets $V_{0,j} = U_{0,j}$ for $1 \le j \le n_1$, and then, for $i = 1, 2, \dots, m$, sets $\hat{\alpha}_i = V_{i-1,(r_i)}$ and $V_{i,j} = (1 - \hat{\alpha}_i)U_{i,j} + \hat{\alpha}_i$ for $1 \le j \le n_{i+1}$. Using a standard result [1, p. 37] on uniform random variables together with an inductive argument, it can be shown that, for $1 \le i \le m$, (1) $F_0^{-1}(V_{i-1,j})$ has the same distribution as $X_{i-1,j}$ for $1 \le j \le n_i$ and (2) $F_0^{-1}(\hat{\alpha}_i)$ has the same distribution as $\hat{\gamma}_i$. In particular, $\bar{F}_0(\hat{\gamma}_m)$ is distributed as $1 - \hat{\alpha}_m$. An easy argument shows that $1 - \hat{\alpha}_i = Z_i(1 - \hat{\alpha}_{i-1})$, where $Z_i = 1 - U_{i-1,(r_i)}$, which implies that $1 - \hat{\alpha}_m = \prod_{i=1}^{m} Z_i$. By construction, the random variables $\{U_{i-1,(r_i)}\}$, and hence the random variables $\{Z_i\}$, are mutually independent. Moreover, each $U_{i,(r_i)}$ has a Beta$(r_i, n_i - r_i + 1)$ distribution [9]. Fixing $m \in \{1, 2, \dots, N\}$, $\eta = (n_1, \dots, n_m)$, and $\pi = (p_1, \dots, p_m)$ satisfying (2)–(3), a calculation using both the independence of the Z_i's and standard properties of the Beta distribution shows that MSRE $= u(\eta, \pi, m)$, where $u(\eta, \pi, m) = h_1(\eta, \pi, m)\big(h_2(\eta, \pi, m)p^{-2} - 2p^{-1}\big) + 1$. Here $h_c(\eta, \pi, m) = \prod_{i=1}^{m}\big((n_i p_i + c)/(n_i + c)\big)$ for $c = 1, 2$.

We now consider how to choose η, π, and m so as to approximately minimize u, subject to (2)–(3). It follows from the results given below that $p \le h_c(\eta, \pi, m) \le 1$ for $c = 1, 2$ and feasible choices of the parameters. Moreover, $h_1(\eta, \pi, m) \approx h_2(\eta, \pi, m)$

for the range of p and N values encountered in practice. Thus, to a good approximation, the problem of minimizing u looks like the problem of minimizing $v(x) = x\big((x/p^2) - (2/p)\big)$ such that $x \in [p, 1]$. It is easy to see that v is strictly increasing on $[p, 1]$, and so we want to make x as small as possible. That is, we want to make h_1 and h_2 as small as possible. To this end, set

$$g_m(N, p, c) = \Big(\frac{(N/m)p^{1/m} + c}{(N/m) + c}\Big)^m$$

for $m \in [1, N]$. We can establish the following result.

THEOREM 1. *For $c \in \{1, 2\}$, $h_c(\eta, \pi, m)$ is minimized by setting $m^*_c = \min\{ m \geq 1 : g_m(N, p, c) < g_{m+1}(N, p, c) \}$, $\eta^*_c = (N/m^*_c, \ldots, N/m^*_c)$, and $\pi^*_c = (p^{1/m^*_c}, \ldots, p^{1/m^*_c})$.*

To summarize, we can approximately minimize the MSRE by (1) computing m^*_1 and m^*_2 as indicated in the theorem, (2) choosing $\theta = \arg\min_{c\in\{1,2\}} u(\eta^*_c, \pi^*_c, m^*_c)$ and setting $m^* = m^*_\theta$, and (3) setting $n_i = N/m^*$ and $p_i = p^{1/m^*}$ for $1 \leq i \leq m^*$. In practice, it is often the case that $m^*_1 = m^*_2$.

Thus we need only choose N. For an MSRE target value ϵ, we can select N by numerically finding $\min\{ N : w(N) \leq \epsilon \}$, where $w(N) = g_{m^*}(N, p, 1)\big(g_{m^*}(N, p, 2)p^{-2} - 2p^{-1}\big) + 1$. Note that $\lim_{N\to\infty} w(N) = 0$, which implies that our quantile estimator converges in mean square to the true value as the total number of DB versions increases.

D. BENCHMARKING

We have implemented a C++ prototype of MCDB-R with the MCDB code base as a starting point, including all of the features described in the paper. As with our MCDB implementation, MCDB-R does not yet have an optimizer or SQL compiler; instead, we use an MCDB-specific language to specify a query plan directly. We now describe a small benchmark of our prototype. Consider the following random version of the TPC-H `orders` table:

```
CREATE TABLE random_ord (o_orderkey, o_yr, o_tot) AS
 FOR EACH o IN (SELECT * FROM orders)
  WITH VAL AS Normal (VALUES(o_mean, o_var))
  SELECT o.o_orderkey, year(o.o_orderdate), v.value
  FROM VAL v
```

This version of `orders` attaches a normally-distributed random "loss" value to each tuple (we use a mean and variance of one). Now, consider the following query:

```
SELECT SUM(val) as totalLoss
FROM random_ord, lineitem
WHERE o_orderkey = l_orderkey AND
   (o_yr = '1994' OR o_yr = '1995')
```

Because the sum of a set of normal variables is also normal, we know that the result of this query must itself have a normal distribution with mean and variance computed as follows:

```
SELECT SUM(grpsize * o_mean) AS mean,
  SUM(grpsize * grpsize * o_var) AS var FROM
(SELECT o_mean, o_var, COUNT(*) AS grpsize
 FROM orders, lineitem
 WHERE year(o_orderdate) in ('1994', '1995')
       AND o_orderkey = l_orderkey
 GROUP BY o_orderkey, o_mean, o_var)
```

Thus, if we run the former query using adding the following:

```
WITH RESULTDISTRIBUTION MONTECARLO(100)
  DOMAIN totalLoss >= QUANTILE(0.999)
```

then we can use the result of the latter query to test the accuracy of the resulting tail sample. Using the TPC-H database (scale-factor $= 10$) on an 8-core server, we ran the former query using

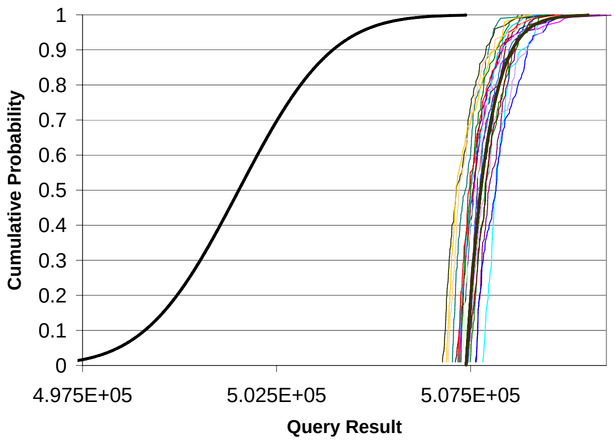

Figure 5: Observed empirical CDFs versus analytic CDF

`GibbsLooper` parameters $m = 5$, $p^{1/m} = 0.25$, $N = 500$, and $l = 100$. This should (if everything is correct) compute a set of 100 samples from the $1.0 - (0.25)^5 = 0.99902 \approx 0.999$ quantile of the query-result distribution. Each TS-seed is used to produce 1000 random values initially. The first three `GibbsLooper` iterations require 156, 124, and 134 seconds to complete. At that point, the first TS-seed runs out of random data, and the query plan is run again to fuel the last two iterations. These iterations take 122 seconds and 115 seconds to run, for a total time of around eleven minutes to complete the tail sampling process and obtain 100 database instances in the upper 0.999-quantile. By comparison, MCDB would require about 18 hours for this task.

Next we tested the accuracy of MCDB-R. We re-ran the above query, but changed how the underlying database is generated to make the example more interesting (and difficult). We modified `random_ord` so that it is generated by selecting 100,000 tuples from the `orders` parameter table. The mean (resp., variance) of each normal used to generate `random_ord` was itself generated by sampling from an inverse gamma-distribution with shape 3 and scale 1 (resp., shape 3 and scale 0.5). The `lineitem` table was generated so that one million of the tuples join with some tuple from `random_ord`, and the rest find no mate. The probability that one of those tuples from `lineitem` will mate with the first of the 100,000 tuples in `random_ord` is $2 \times 10^{-5} - 10^{-10}$. The probability that the tuple will mate with the ith tuple in `random_ord` is equal to the probability that it will mate with the $(i - 1)$th tuple, minus $2 \times (10^{-5} - 10^{-10})/(10^5 - 1)$.

We ran MCDB-R 20 times on this query using `GibbsLooper` parameters $m = 5$, $p^{1/m} = 0.25$, $N = 1000$, and $l = 100$. Figure 5 shows the 20 empirical tail CDFs that we observed (each based on 100 samples). The true CDF for the query-result distribution (calculated analytically as described above) appears as a thick black line on the left side of the plot. The true tail CDF corresponding to the 0.99902 quantile of the query-result distribution appears as a thick black line on the right.

It is clear from the plot that the 20 empirical tail CDFs cluster closely around the true tail CDF. We also obtained 20 estimates of the 0.99902 quantile by recording the value of the minimum tail sample in each of the 20 runs. The mean value of these 20 estimates was 5.0728e5, whereas the true quantile value was 5.0738e5. The empirical standard error of the estimates was 265. To put this in perspective, the middle 99% of the query-result distribution has a width of approximately 2503; i.e., by generating only 1000 intermediate database instances (efficiently, using Gibbs tuples), MCDB-R is able to "walk" out to the upper 0.99902 quartile and incur a standard error of only 10%.